Departamento de Tecnoloía Electrónica

TRABAJO FIN DE GRADO

Design and Implementation of a Multi-Level Monte Carlo Accelerator for Option Pricing on the Zynq-7000 EPP

Autor: Helena Lázaro García

Director: José Antonio García Souto
Tutor: Celia Lopez Óngil

Leganés, Febrero 2013

Ttulo: Design and Implementation of a Multi-Level Monte Carlo Accelerator for Option Pricing on the Zynq-7000 EPP
Autor: Helena Lázaro García
Director: José Antonio García Souto
Tutor: Celia Lopez Óngil

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el da ..... de ..... de ... en ....., en la Escuela Politcnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIN de:

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Agradezco a mi tutora, Celia, principalmente, debido a que gracias a ella pude llevar este proyecto fin de grado en el extranjero con su supervisión desde España. También me gustaría agradecer a la Universidad de Kaiserslautern que me aceptó como free mover y siempre tuvo sus puertas abiertas para todo lo que necesitara, así como mis co-tutores en el extranjero Christian y el profesor Wehn, que me brindaron y facilitaron en la medida de lo posible el entrar a formar parte de su equipo de investigación y a trabajar con ellos. Por último, pero no menos importante, todo el apoyo y ánimo que me ha aportado mi familia durante todo el proceso.

Abstract

Current trends in financial modeling aim to predict the market prices of different financial instruments, specially derivatives as Options. These models and prediction algorithms are run in server farms in order to meet the demand of data processing. As a result an increase on the number of such processing compounds has arisen, at a point that there is not enough energy to supply them. In this work we present a novel concept for High Performance Computing, which comprises the use of Embedded Systems Theory, specially the use of FPGA based hardware, in order to accelerate computing algorithms. To the best of our knowledge, we implement the first hardware accelerator of the Heston Model in a leading-edge technology development board (Zynq EPP).

Tendencias actuales en modelos de financieros intentan predecir los precios de mercado de distintos instrumentos financieros, especialmente en derivados de opciones. Estos modelos y algoritmos predictivos se corren sobre granjas de servidores para alcanzar la demanda del procesamiento de datos. El resultado obtenido ha sido un gran incremento del procesamiento, hasta el punto de no tener suficiente energía para alimentarlo. En este trabajo se presenta un nuevo concepto de alto rendimiento computacional, el cual comprende el uso de teoría de sistemas empotrados, especialmente el uso de FPGAs basadas en hardware, con el fin de acelerar algoritmos computacionales. Con lo mejor de nuestro conocimiento, hemos implementado el primer acelerador de hardware basado en el modelo de Heston en una tarjeta de desarrollo de tecnologia punta (Zynq EPP).

Keywords: FPGA, Hardware, Accelerator, Zynq.

Contents

List of Figures

List of Tables

1. Introduction

Modern financial mathematics consume more and more computational power and energy. Finding efficient algorithms and implementations to accelerate calculations is therefore a very active area of research. In the University of Kaiserslautern, Germany, an interdisciplinary cooperation rose up between the Microelectronic System Design Research Group (`http://ems.eit.uni-kl.de` and the Stochastics and Financial Mathematics Department (`http://www.mathematik.uni-kl.de/~wwwfm/index_eng.html`) to try to build optimal designs.

Today, pricing of derivates (particularly options) in financial institutions is a challenge. Besides the increasing complexity of the products, obtaining fair prices requires more realistic (and therefore complex) models of the underlying asset behavior. Not only due to the increasing costs, energy efficient and accurate pricing of these models becomes more and more important

In the University of Kaiserslautern they are currently designing a highly parallel architecture for field programmable gate arrays based on the multi-level Monte Carlo method. It is optimized for high throughput and low energy consumption, compared to GPGPUs.

My work throughout this thesis will be the migration of the previous hardware accelerator developed by Pedro Torruela[3] in his Master thesis to the ZedBoard for the purpose of opening the doors to the finally implementation of a *hardware-software-co-design flow*.

Thanks to the Zynq-7000, that combines a high performance multicore processing subsystem ARM-based with FPGA structure, we are able to migrate the complete previous hardware accelerator to the new board, integrating also the host inside the board. For that we will install an operative system (Linux distribution) in the board.

2. Background

In this section we will address general concepts needed in order to understand the next chapters of this work. We begin by briefly explaining some of this concepts and finish by making a quick summary of the previous work.

2.0.1 Option Pricing

The pricing of options is a very important problem encountered in financial markets today. Fisher Black and Myron Scholes developed in 1973 a method to determine the value of derivates [10]. This method was referred in the later on publication of Robert C. Merton, "Theory of Rational Option Pricing "[11]. This model was called Black-Scholes and provides a conceptual framework for valuing options, such as calls or puts. But for many other options, either there are no closed from solutions, or if such closed form solutions exist, the formulas exhibiting them are complicated and difficult to evaluate accurately by conventional methods. Monte Carlo methods provide the highest flexibility in application and are very robust, therefore we go for MC methods in this work.

2.1 Multi-Level Monte Carlo Simulation

The name Monte Carlo was applied to a class of mathematical methods first used by scientists (Ulam and Metropolis) working on te development of nuclear weapons in Los Alamos in 1940s [8]. The essence of the method is the invention of games of chance whose behavior and outcome can be used to study some interesting phenomena, such as a new product's sales or stock prices (in the financial area). It is, in essence, a computerized mathematical technique that allows people to account for risk in quantitative analysis and decision making.

Monte Carlo simulation offers an alternative to analytical mathematics for understanding a statistic's sampling distribution and evaluating its behavior in random samples. Monte Carlo simulation does this empirically using random samples from known populations of simulated data to track a statistic's behavior. The basic concept is straightforward: If a statistic's sampling distribution is the density function of the values it could take on in a given population, then its estimate is the relative frequency distribution of the values of that statistic that were actually observed in many samples drawn from that population. Because it usually is impractical for social scientists to sample actual data multiple times, we use artificially generated data that resemble the real thing in relevant

ways. The recent availability of high-speed computers makes this approach now widely practical.

The gambling analogy notwithstanding, Monte Carlo simulation is a legitimate and widely used technique for dealing with uncertainty in many aspects of business operations. For our purposes, it has been shown to be an accurate method of pricing options and particularly useful for path-dependent options and others for which no known formula exists.

In 2001 Stefan Heinrich [7] was the first who study MC approximations to high dimensional parameter dependen integrals. Heinrich surveyed the multi-level variance reduction technique introduced by himself and presented extensions and new developments of it. Seven years later, Giles introduced multi-level Monte Carlo path simulation method [6] where he shows that multigrid ideas can be used to reduce the computational complexity of estimating an expected value arising from a stochastic differential equation using MC path simulations. Although the details of both multi-level Monte Carlo methods are quiet different, the analysis of the computational complexity is quiet similar.

In its most general form, multi-level Monte Carlo (MLMC) simulation uses a number of levels of resolution, $l = 0, 1, ..., L$, with $l = 0$ being the coarsest, and $l = L$ being the finest. In the context of a SDE simulation, level 0 may have just one timestep for the whole time interval $[0, T]$, whereas level L might have 2^L uniform timesteps $\triangle t_L = 2^{-L}T$. This improves the computational efficiency of Monte Carlo path simulation by combining results using different numbers of timesteps.

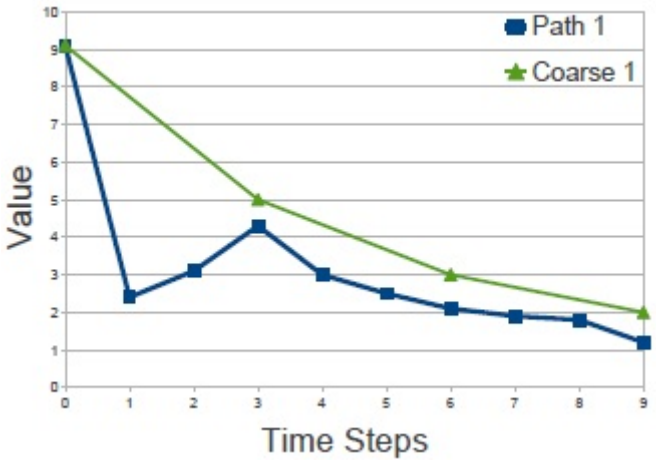

Figure 2.1: Hypothetical evolution of a multi-level Monte Carlo Simulation

Summarizing, in the multi-level Monte Carlo Simulations, the Brownian or random values used to compute the next value of a path, are added for a certain number of loops, and then used to compute a coarse next value that will be used

later on in the final calculation [4]. The use of this approach has demonstrated to reduce the complexity of Monte Carlo Path Simulations [Giles, 2008]. Figure 2.1 shows the evolution of an hypotethical multi-level Monte Carlo Simulation. Notice that the path in green only has 4 information points, that correspond to the evolution of the path every three fine steps.

2.2 Zynq-7000 EPP - An Extensible Processing Platform Family

Zynq-7000 All Programmable SoCs (AP SoC) are extensible processing platforms, a new generation of SoCs that combine high performance multicore processing subsystems ARM-based with FPGA structure.

The Zynq-7000 family combines an industry-standard ARM dual-core Cortex(TM) -A9 MPCore(TM) processing system with Xilinx's scalable 28nm programmable logic architecture. It supports parallel development of software for the dual-core Cortex-A9 processor-based system and custom accelerators and peripherals in the programmable logic. The ARM Cortex-A9 CPUs are the heart of the PS and also include on-chip memory, external memory interfaces, and a rich set of peripheral connectivity interfaces. Software developers can leverage the open source Eclipse platform, Xilinx Platform Studio Software Development Kit (SDK), ARM Development Studio 5 (DS-5) and ARM RealView Development Suite (RVDS), or compilers, debuggers, and applications from different vendors.

The flexible nature of programmable logic and its tight integration to the ARM based processing system offers the possibility to add virtually any peripheral we want and create accelerators to extend the performance of the Zynq-7000 devices. This makes Zynq-7000 devices the ideal solution for our work.

The Zynq-7000 architecture enables implementation of custom logic in the PL and custom software in the PS. It allows for the realization of unique and differentiated system functions. The integration of the PS with the PL allows levels of performance that two-chip solutions (e.g., an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency, and power budgets.

The inclusion of an application processor enables high-level operating system support, e.g., Linux. Other standard operating systems used with the Cortex-A9 processor are also available for the Zynq-7000 family.

The PS and the PL are on separate power domains, enabling the user of these devices to power down the PL for power management if required. The processors in the PS always boot first, allowing a software centric approach for PL configuration. PL configuration is managed by software running on the CPU, so it boots similar to an ASSP.

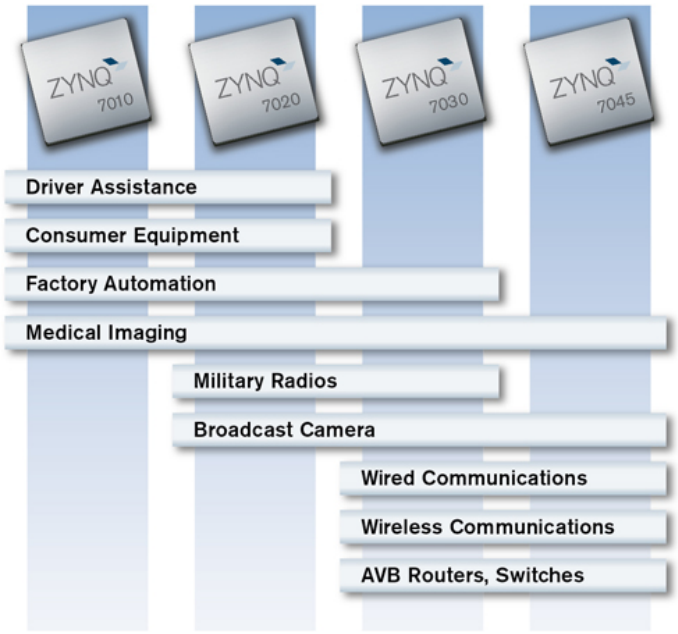

Figure 2.2: Zynq 7000 family

2.2.1 ZedBoard - Hardware Development Platform

The ZedBoard (`http://www.zedboard.org`) enables hardware and software developers to create or evaluate Zynq-7000 All Programmable SoC designs. Nowadays it is available for purchase from Avnet Electronics Marketing (`http://www.avnet.com/`) and Digilent (`http://www.digilentinc.com/`).

The expandability features of this evaluation and development platform make it ideal for rapid prototyping and proof-of-concept development. The ZedBoard includes Xilinx XADC, FMC (FPGA Mezzanine Card), and Digilent Pmod compatible expansion headers as well as many common features used in system design. ZedBoard enables embedded computing capability by using DDR3 memory, Flash memory, gigabit Ethernet, general purpose I/O, and UART technologies.

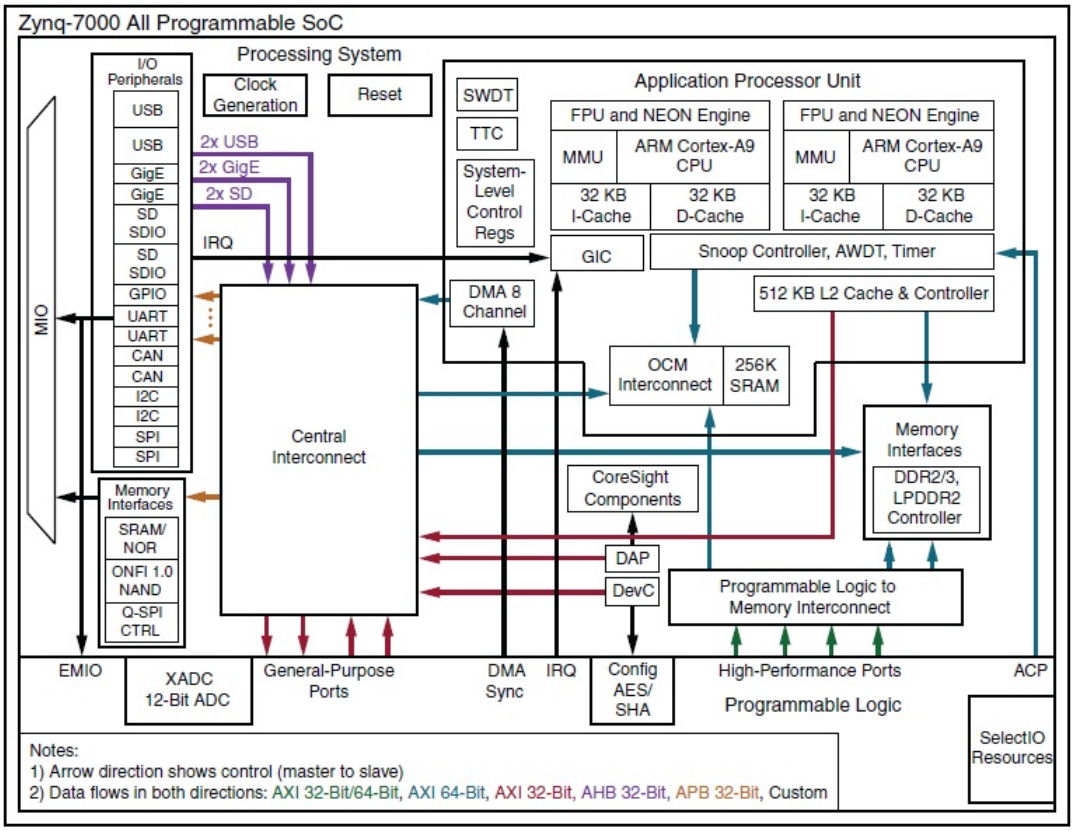

Figure 2.3: Functional blocks of the Zynq-7000 All Programmable SoC

2.3 Xillybus - IP Cores and Design Services

Xillybus (`http://xillybus.com/`) is a IP core developed by Xillybus Ltd. company. The downloaded evaluation core is free for any use, as long as this use reasonably matches the term "evaluation". This includes incorporation the core in end-user designs, running real-life data and field testing. It is a straightforward, intuitive, efficient DMAbased end-to-end turnkey solution for data transport between an FPGA and a host running Linux or Microsoft Windows. It's available for personal computers and embedded systems using the PCI Express bus as the underlying transport, as well as ARM-based processors, interfacing with the AMBA bus (AXI3/AXI4).

The FPGA designer as well as the host application programmer interact with Xillybus through well-known interfaces: The FPGA application logic connects to the IP core through standard FIFOs; the user application on the host performs plain file I/O operations on pipe-like device files. Streaming data moves naturally between the FIFO and the file handler opened by the host application. There is no specific API involved.

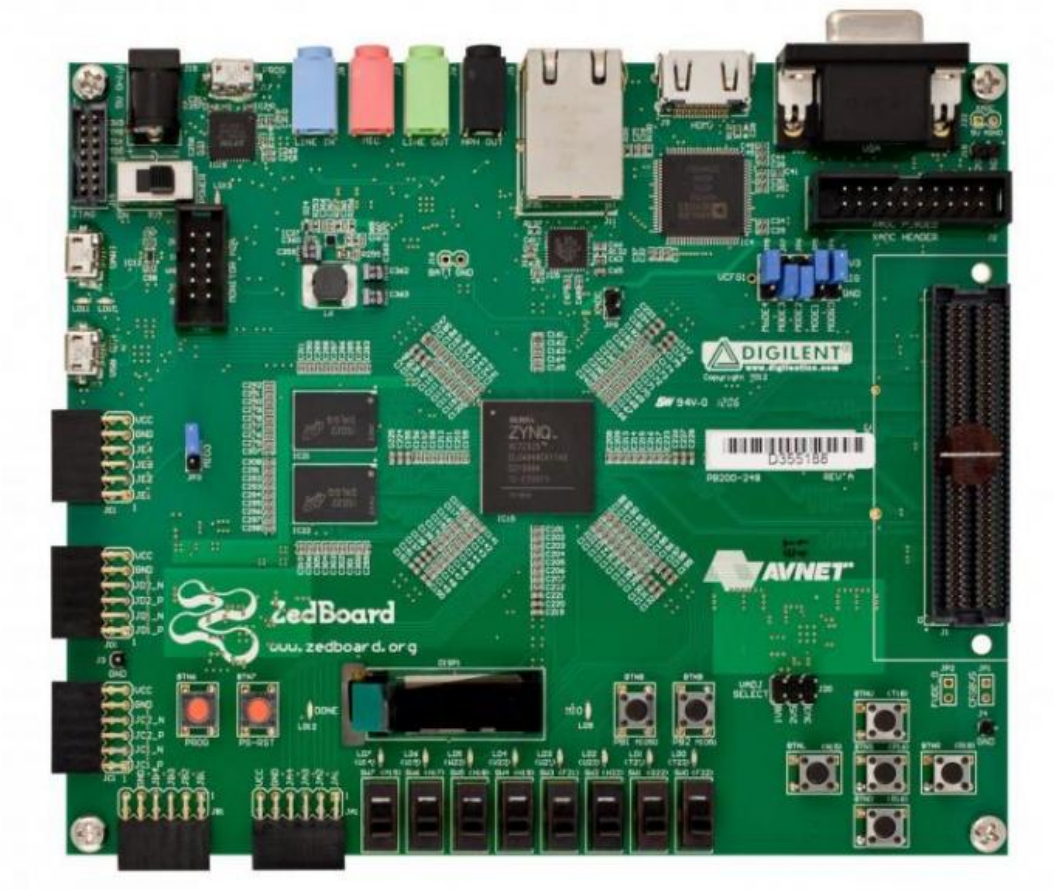

Figure 2.4: ZedBoard

For example, writing data to the lower FIFO in the diagram above makes the Xillybus IP core sense that data is available for transmission in the FIFOs other end. Soon, the Xillybus reads the data from the FIFO and sends it to the host, making it readable by the user-space software. The data transport mechanism is transparent to the application logic in the FPGA, which merely interacts with the FIFO.

On its other side, the Xillybus IP core implements the data flow utilizing the AXI bus, generating DMA requests on the processor core's bus.

The Xillybus IP core communicates data with the user logic through a standard FIFO ("Application FIFO" in the diagram), which is supplied by the IP core's user. This gives the FPGA designer the freedom to decide the FIFO's depth and its interface with the application logic. This setting relieves the FPGA designer completely from managing the data traffic with the host. Rather, the Xillybus core checks the FIFOs "empty" or "full" signals (depending on data direction), and initiates data transfers when the FIFO is ready for it.

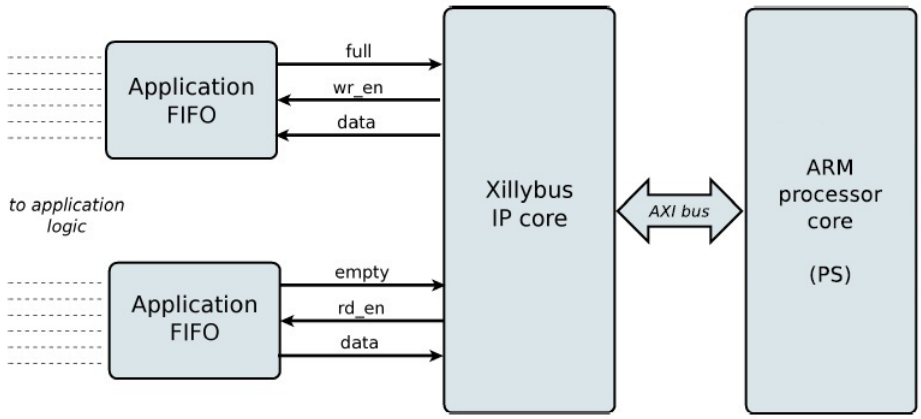

Figure 2.5: Simplified FPGA block diagram of Xillybus

There is no kernel-space or hardware-related programming necessary on the host side, nor any need to link with a particular software library. Any practical programming language can be used to access Xillybus streams without a specific extension.

The host driver generates device files that behave like named pipes: They are opened, read from and written to just like any file, but behave much like pipes between processes or TCP/IP streams. And like the TCP/IP socket, the Xillybus stream is designed to work well with high-rate data transfers as well as single bytes arriving or sent occasionally.

2.4 Xillinux - A Linux Distribution for the Zedboard

Xillinux is a complete, graphical, Ubuntu 12.04 LTS-based Linux distribution for the Zedboard, intended as a platform for rapid development of mixed software / logic projects. It is developed by Xillybus Ltd. company and provided in its web page. It is released under GPL and can be used with no restriction like any Linux distribution such as RHEL, Fedora, Ubuntu, etc. Xillinux is a collection of software which supports roughly the same capabilities as a personal desktop computer running Linux. Unlike common Linux distributions, Xillinux also includes some of the hardware logic, in particular the VGA adapter.

The distribution is organized for a classic keyboard, mouse and monitor setting. It also allows command-line control from the USB UART port, but this feature is made available mostly for solving problems.

Xillinux is also a kick-start development platform for integration between the device's FPGA logic fabric and plain user space applications running on the ARM processors. With its included Xillybus IP core and driver, no more than basic programming skills and logic design capabilities are needed to complete

the design of an application where FPGA logic and Linux-based software work together.

The bundled Xillybus IP cores eliminates the need to deal with the low-level internals of kernel programming and interface with the processor, by presenting a simple and yet efcient working environment to the application designers.

2.5 Previous Work

Nowadays the modern financial mathematics is running fast and the energy and power consumption are becoming an important issue to approach. An interdisciplinary cooperation rise up between the Microelectronic Systems Design Research Group (`http://ems.eit.uni-kl.de/`) and the Stochastic and Financial Mathematics Department (`http://www.mathematik.uni-kl.de/~wwwfm/index_eng.html`) in the University of Kaiserslautern, Germany. The main goal of this cooperation is finding efficient algorithms for an implementation on dedicated hardware.

In the autumn 2012 Pedro Torruellal[3] proposed an implementation for a FPGA based multi-level Monte-Carlo Hardware Accelerator for the Heston Mode basing his study on the previous energy efficient model proposed in the International Conference on Reconfigurable Computing and FPGAs [9]. Pedro's work is the first known implementation of hardware accelerator for multi-level Monte Carlo Simulation on a FPGA. The main goal of his work was to have a working platform and framework, with which we can continue further activities.

The image 2.6 summarizes hardware accelerator that was developed. The blocks inside the light yellow rectangle represents the host computer, here is where a high-level software program is running, and is in charge of doing the payout computation to control the Monte Carlo Simulation. In the embedded platform, the big second block in light purple, the hardware accelerator is connected to CPU. This CPU in this application is a Xilinx MicroBlaze, synthesized together with the accelerator in a Virtex-6 FPGA. The accelerator is implementing directly part of the algorithm of the Monte Carlo simulation.

This accelerator will be synthesized to support a clock frequency of 100 Mhz along with the MicroBlaze. The data representation in this accelerator will be IEEE-754 single precision floating point standard. In general, the datapath will have a bit width of 66 bits: 32 for the price and volatility values, and 2 more for hit barrier status flags. It is important to maintain only valid packets inside the datapath and it is also important to maintain their ordering. With this assumption we were able to avoid the use of a packet identification word in the architecture. This is critical in order to save resources on the FPGA, that otherwise would just be transporting this constant signals without doing any computation with them.

The data inside the datapath is handled with a packet philosophy, where each fine and coarse path is represented by a packet, having a volatility-price pair

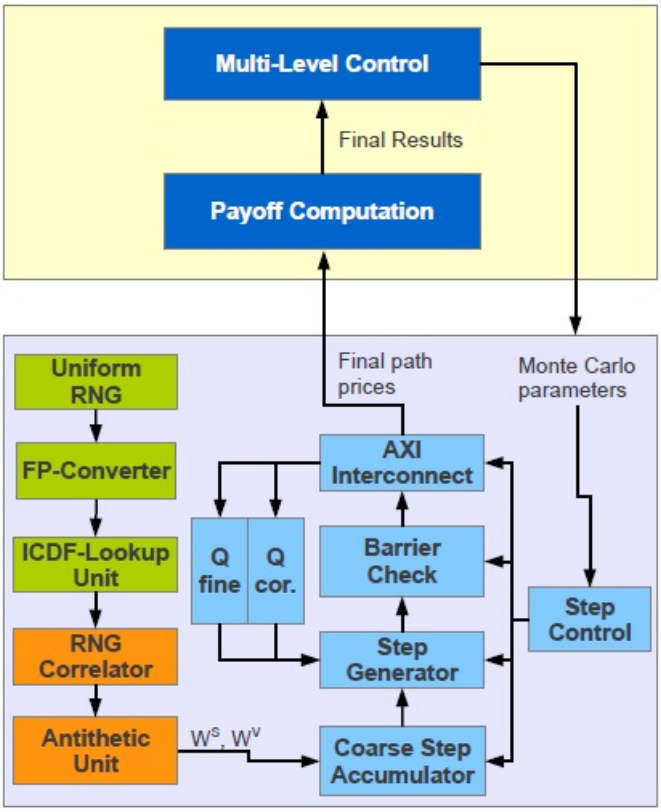

Figure 2.6: Block Diagram of the System's Architecture. In dark blue: software running in a host machine. In green: random number generation cores. In orange: accelerator's external datapath. In light blue: accelerator's internal datapath.

and the barrier bits. Figure 2.7 shows this packet structure. In this example we have six fine paths and their corresponding coarse path. The fine paths are evolved several times before the coarse paths.

In general the architecture of the accelerator is pipelined with a cyclic inner datapath that allows the direct feedback of data packets for the iterations of the Monte Carlo Simulation. This is represented in Figure 2.6 as the blocks inside the grey rectangle in the bottom. As you can see there, the accelerating logic has two parts: the modules that feed random numbers into the datapath (in green and orange) and the application's datapath (in light blue), which is based upon different cores that implement a part of the algorithm.

From this point, the use of a *hardware-software-co-design flow* is a need in order to find the best possible configuration of software and hardware blocks. This means the complete migration of the computation to the embedded platform could be carried out. In the actual accelerator's state the CPU only serves as an interface between the data generated by the accelerator and the external

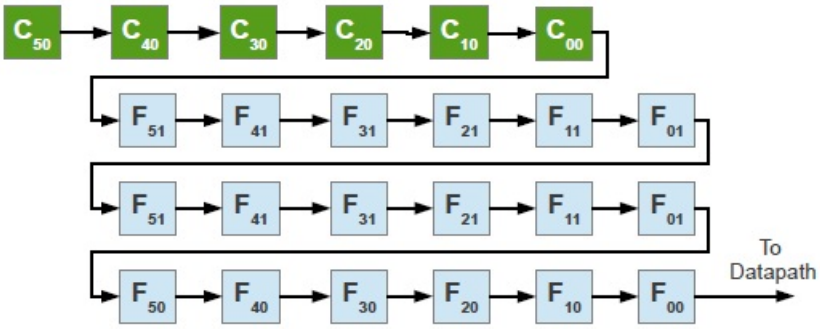

Figure 2.7: Graphical representation of the packets queue in the system. The first subscript represents the id of the path, the second represents the iteration number.

host computer, so we are not exploiting our resources. At this point is where the use of the Xilinx's Zynq EPP is interesting, since we could use the processing power of the ARM core to run interfacing and computing tasks.

My work throughout this thesis will be the migration of the previous hardware accelerator to the ZedBoard for the purpose of opening the doors to the finally implementation of a *hardware-software-co-design flow*. Afterwards there will be no need to use a host computer, since we will implement a completely independent system while installing an operative system (Linux distribution) in the ZedBoard.

3. Implementation

In this section we aim to describe the general implementation procedure of the system. Throughout this section we will first prepare the environment for the hardware accelerator building Xillinux in the ZedBoard. At this point we will be able to boot up the Xillybus project and test that it works properly testing FIFO's functionality and making small modifications over them. Later on we will proceed to integrate our own custom logic. At the end we will have the hardware accelerators working in the new platform, the Zynq.

Our aim is to eliminate the need of a host computer, since we will have it implemented on our ZedBoard. We have summarized our work in the Image 3.1.

The flow of our work has different steps, some only need to be done once (setting up the operative system, booting up for first time the proyect Xillybus), since others take several iterations to get the desired result (IP core development, emulation of the results). We can appreciate it in the Figure 3.2

3.1 Setting up Xillinux

The first step we have to carry out is building Xillinux distribution in the ZedBoard.

The Xillinux distribution is intended as a development platform: A ready-for-use environment for custom logic development and integration is built during its preparation for running on hardware. This makes the preparation for the first test run somewhat timely but significantly shortens the cycle for integrating custom logic.

To boot the Xillinux distribution from an SD card, it must have two components:

- An initial boot image environment, consisting of boot loaders, a configuration bitstream for the logic fabric (known as PL), and the binaries for booting the Linux kernel.

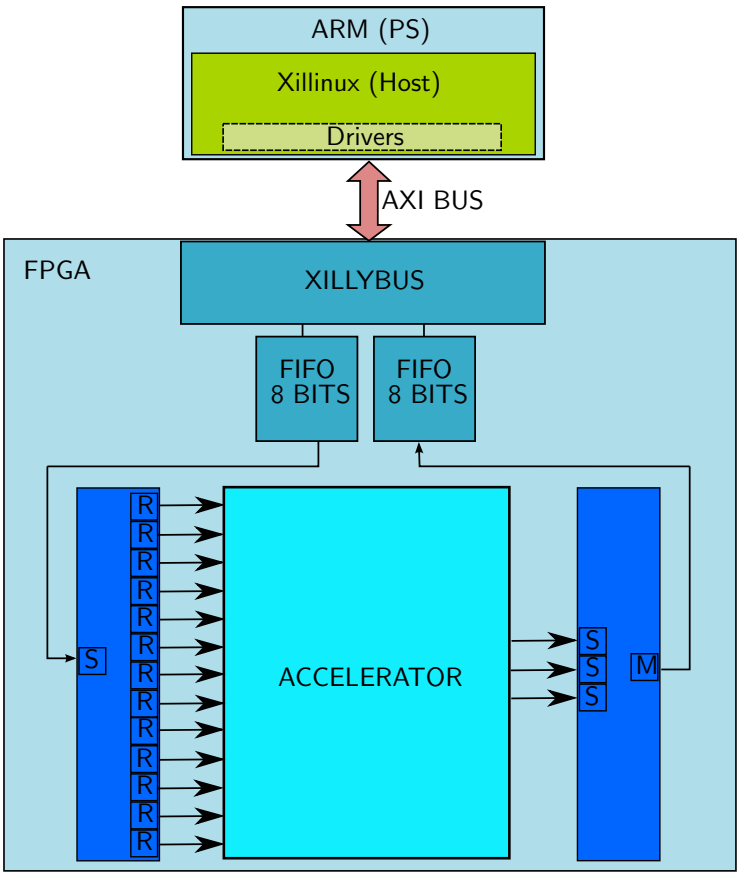

Figure 3.1: Hardware and Software implementation. The Hardware part is represented in blue and the Software part in green.

- A root file system mounted by Linux.

The following steps must be done in the order outlined below:

- Unzipping the boot image kit.
- Generating the processor netlist.
- Generating Xilinx IP cores.
- Implementing the main logic fabric project.
- Producing a boot image file.
- Writing the raw Xillinux image to the SD card.
- Copying the boot image file into the SD card.

We obtain the needed files from the Xillybus web page(`www.xillybus.com`).

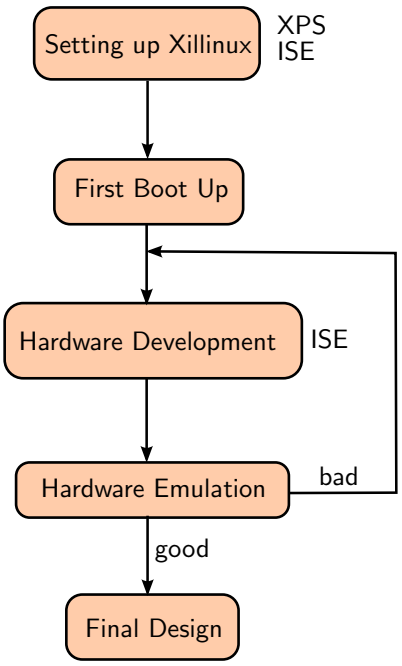

Figure 3.2: General block diagram of the work flow followed in this project. Near each process appears the tools we use for this section, for Xilinx tools we use the Version 14.2.

3.1.1 Unzipping the boot image kit

The first step is unzipping the previously downloaded xillinux−eval−zedboard−XXX.zip file into a working directory.

The bundle consists of the following directories:

- verilog - Contains the project file for the main logic and some sources in Verilog (in the src subdirectory)
- vhdl Contains the project file for the main logic and some sources, with the user-editable source file in VHDL (in the src subdirectory)
- cores Precompiled binaries of the Xillybus IP cores
- system Directory for generating processor-related logic
- runonce Directory for generating general-purpose logic (CoreGen FIFO IP cores).
- boot Final stage assembly of the boot image file.

The interface with the Xillybus IP core takes place in the xillydemo.v or xillydemo.vhd files in the respective src subdirectories. This is the file to edit in order to try Xillybus with our own data sources and sinks.

3.1.2 Generating the processor netlist

To generate the processor netlist we must launch the Xilinx Platform Studio (XPS) and open the system.xmp file, wich is inside the "system" directory. Once the project is opened we must click "Generate Netlist" to the left. We will use the version 14.2 of XPS.

The console output says "XST completed" and "Done!" upon a successful completion of the process. At this pint, close the XPS completely.

Generating Xilinx IP cores

At this point we are going to rebuilt the Xilinx IP cores:

- fifo_32x512
- fifo_8x2048
- vga_fifo

Within the boot image kit, double-click the runonce.xise file in the "runonce" directory. This opens the Xilinx ISE Project Navigator. On the opened window we have the three cores we need to regenerate. We are going to click on by one and run "Regenerate Core" (we find this option expanding the "CORE Generator" line in the process window).

At this point we close the ISE Project Navigator completely.

3.1.3 Generating the bitstream file

Here we can use two projects, the only difference is that one is written in VHDL and the other in Verilog. For our project we are going to use the VHDL one. Taking into account that, we are going to use the project saved in the directory "vhdl".

The Project Navigator will launch and open the project with the correct settings. We just need to click "Generate Programming File".

The procedure will produce several warnings (FPGA implementations always do) but should not present any errors. The process should end with the output Process Generate Programming File completed successfully. The result can be found as xillydemo.bit in the "vhdl" directory along with several other files.

When this task is completed successfully, we close the ISE Project Navigator completely.

3.1.4 Creating the boot.bin image

To create the boot.bin image we need the xillydemo.bit file created before. We will copy this file from the "vhdl" subdirectory into the "boot" directory.

Now we need a terminal open in this directory. As we are working over a Linux OS, to be able to carry out the next task we must run the configuration settings of ISE tools'. After this we can run the following command:

```
# bootgen -image xillybus.bif -o i boot.bin
```

This command will create the boot.bin image that we need for the SD card.

3.1.5 Loading the image (Linux)

First of all, it is important to detect the correct device as the SD card. This is best done by plugging in the USB connector and looking for something like this in the main log file:

In our case the name the kernel gave to the new disk is "sdb" in the example above.

When we know which is our disk, we proceed to uncompress the image file.

```
# gunzip xillinux.img.gz
```

Afterwards we must copy the image to the SD card.

```
# dd if=xillinux.img of=/dev/sdb bs=512
```

To finish we will verify is the content of the image is the same as the content of the SD card.

```
# cmp xillinux.img /dev/sdb
cmp: EOF on xillinux.img
```

Note the response: The fact that EOF was reached on the image file means that everything else compared correctly, and that the flash has more space than actually used. If cmp says nothing (which would normally be considered good) it actually means something is wrong. Most likely, a regular file /dev/sdc was generated rather than writing to the device.

If the process has been completed successfully we should have an SD card with two partitions.

Before continuing with the next steps we should unmount the SD card.

3.1.6 Copying the boot image file into the SD card

Connect the SD card back to the computer. Then copy the boot.ini to the FAT file system on the SD card, it is the first (and smaller) partition. This partition only has two files: "devicetree.dtb" and "zImage".

To finish we unmount the SD card properly. Now we can introduce the SD card in the ZedBoard and start with the booting up.

3.2 Booting up Xillybus

In this section we will set up the ZedBoard for the booting up.

3.2.1 Jumper settings

For the board to boot from the SD card, modifications in the jumper settings need to be made. The correct setting is depicted in the image 3.3. The following jumper changes are necessary:

- Install a jumper for JP2 to supply 5V to USB device.
- JP10 and JP9 moved from GND to 3V3 position, the three others in that row are left connected to GND.
- Install a the jumper for JP6.

3.2.2 Attaching peripherals

The following general-purpose hardware should be attached the board:

- A computer monitor to the analog VGA connector. Since Xillinux produces a VESA-compliant 1024x768 @ 60Hz through the analog VGA plug, it's almost certain that any computer monitor will suffice.
- A mouse and keyboard to the USB OTG connector, through the USB female cable that came with the board (which is also the shorter one). The system will boot in the absence of these, and there is no problem connecting and disconnecting the keyboard and mouse as the system runs the system detects and works with whatever keyboard and mouse it has connected at any given moment. Note that JP2 on the board must be installed for this USB port to function.
- The Ethernet port is optional for common network tasks. The Linux machine configures the network automatically if the attached network has a DHCP server.
- The UART USB port is optionally connected to a PC, but is redundant in most cases. Some of the boot messages are sent there, and a shell prompt is issued on this interface when the boot completes. This is useful when either a PC monitor or a keyboard is missing or don't work properly.

3.2.3 Powering up the board

This paragraph describes what to expect on a proper cycle from powering up the system.

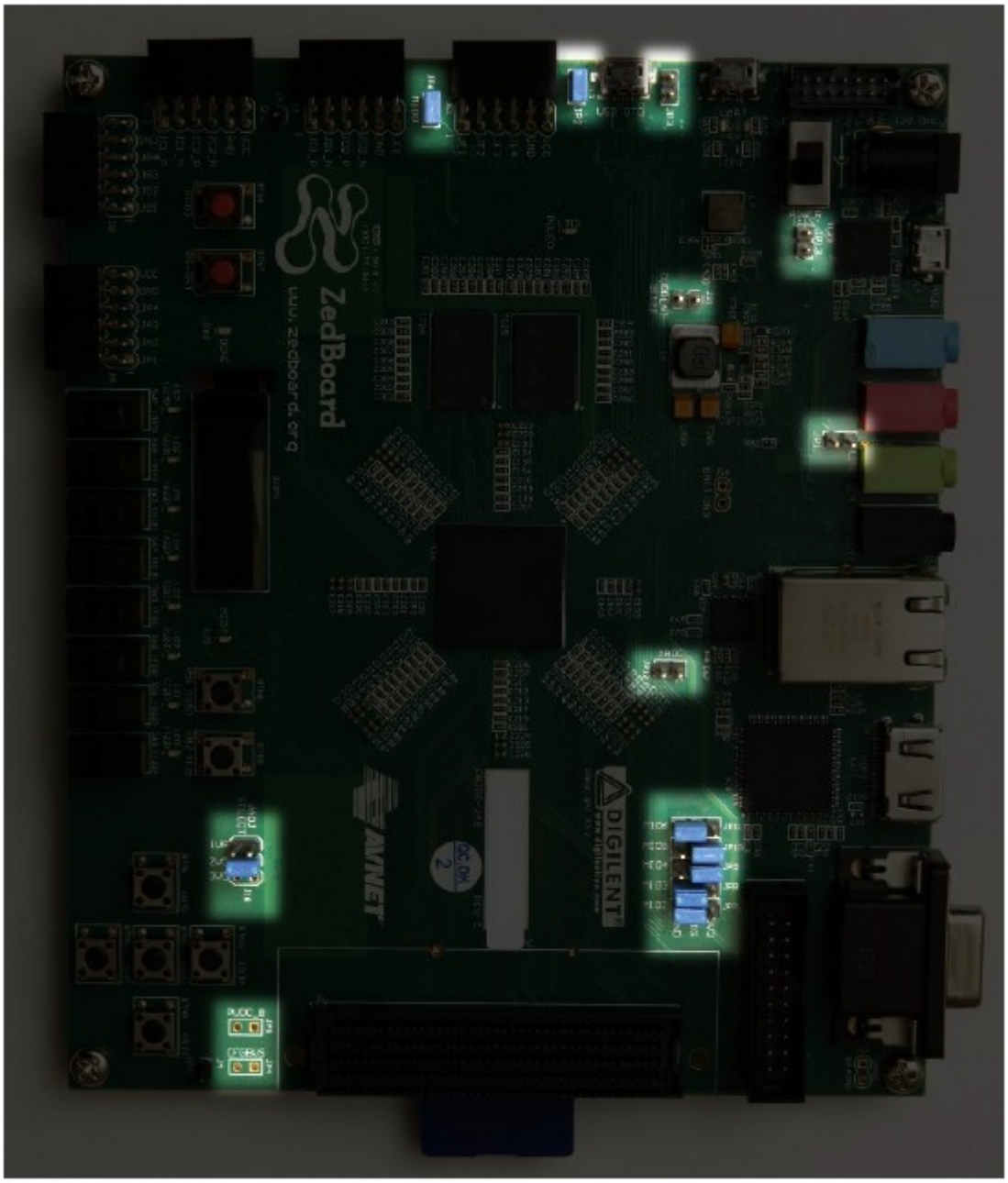

Figure 3.3: Jumper settings highlighted on the Zedboard

Plug the SD card into the Zedboard, and power it on. The following sequence is expected:

- Only the green POWER LED goes on. Nothing else happens.
- About 8 seconds later, the blue DONE LED goes on, and a red LED starts blinking. Other LEDs go on as well. The VGA monitor displays a screen saver pattern with Xillybus logo moving on white background. All these indicate that the logic fabric (PL, FPGA) has been loaded properly with the bitstream (xillydemo.bit).
- After some additional 14 seconds (22 seconds from power-up), Linux boot up text appears rapidly on the VGA monitor. This looks exactly like a PC booting, with white text on black background.
- A login prompt should appear no later than 10 seconds after the boot up text was first seen on the VGA monitor. The system auto-logins as root, presenting a greeting message and a shell prompt. A similar shell prompt is also presented at the USB UART link, mostly for troubleshooting.

Type startx at command prompt to launch a Gnome graphical desktop. The desktop takes some 15-30 seconds to initialize. If nothing appears to happen, monitoring the activity meter on the OLED display helps telling if something is going on.

3.2.4 Allow remote SSH access

Allowing remote SSH access is not a need but is very useful.

To install an ssh server on the board, connect the board to the Internet and type:

```
# apt-get install ssh-server
```

at shell prompt. Please note that the root password is none by default, and ssh rightfully refuses to login someone without a password.

To rectify this, set the root password with

```
# passwd root
```

at shell prompt.

3.2.5 Shutting down

To power down the system, pick the top-right icon on the desktop, and click Shut Down.... Alternatively, type

```
# shutdown -h now
```

at shell prompt.

When a textual message saying System Halted appears, its safe to power the board off.

3.2.6 Taking it from here

Our Zedboard has now become a computer running Linux for all purposes. From now on we will interact with the logic through the Xillybus IP core. Note that the driver for Xillybus is already installed in the Xillinux distribution.

Xillinux includes the gcc compiler and GNU make, so host applications can be compiled natively on the boards processors. Additional packages may be added to the distribution with apt-get as well.

3.3 Hardware Development

The Xillinux distribution is set up for easy integration with application logic. The front end for connecting data sources and sinks is the xillydemo.vhd file. All other HDL files in the boot image kit can be ignored for the purpose of using the Xillybus IP core as a transport of data between the Linux host and the logic fabric.

Additional HDL files with custom logic designs may be added to the project regenerating the bitstream file (see paragraph 3.1.3), and then rebuilt the same way it was done the first place. To boot the system with the updated logic, the boot.bin needs to be regenerated as described in paragraph 3.1.4 and written to the SD file as described in paragraph 3.1.6. There is no need to repeat the other steps of the initial distribution deployment, so the development cycle for logic is fairly quick and simple.

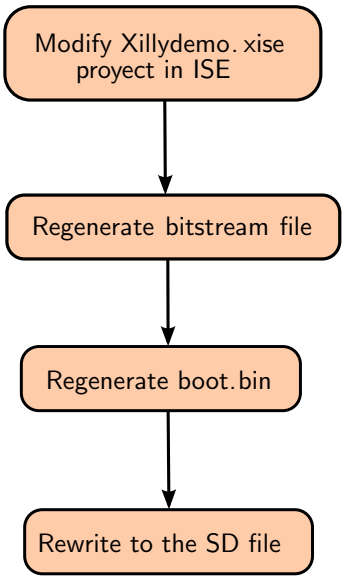

Figure 3.4: Hardware development work flow for new custom logic integration.

3.3.1 FIFOs Overview

These FIFOs have been generated with the *Core Generator* function from the ISE tool. The Xilinx LogiCORE IP FIFO Generator[2] is a fully verified first-in first-out (FIFO) memory queue for applications requiring in-order storage and retrieval. The core provides an optimized solution for all FIFO configurations and delivers maximum performance (up to 500 MHz) while utilizing minimum resources.

The FIFO Generator core supports Native interface FIFOs and AXI4 interface FIFOs. The Native interface FIFO cores include the original standard FIFO functions delivered by the previous versions of the FIFO Generator[1] (up to v6.2). Native interface FIFO cores are optimized for buffering, data width conversion and clock domain decoupling applications, providing in-order storage and retrieval.

3.3.2 Testing initial FIFOs' funcionality

The idea about this test is to verify that the loopback indeed works. The easiest way is by using the UNIX command-line utility cat.

We open two terminal windows, in the first one we type at command prompt:

```
# cat /dev/xillybus_read_8
```

On the second terminal window we type:

```
$ cat > /dev/xillybus_write_8
```

In the first windows, the "cat" program will print out anything it reads from the xillybus_read_8 device file, meanwhile in the second window we should notice the > redirection sign, which tells "cat" to send anything typed to xillybus_write_8. We can see it running in the Figure 3.5.

Now, whenever we type some text on the second terminal, and press ENTER, the same text will appear in the first terminal. The reason nothing is sent until ENTER is pressed, is that the standard input is designed not to bother applications with every character.

Either "cat" can be halted with a CTRL-C. Other trivial file operations will work likewise

3.3.3 Adding a second FIFO

Before adding our own HDL files we add a second FIFO in a loopback. Instead of having just one FIFO of 8 bits we modified the xillydemo.vhd file in order to add a second one connected to the first and afterwards, connect this one to the Xillybus. We can see the connexion between the FIFOs and the Xillybus in the Figure 3.6.

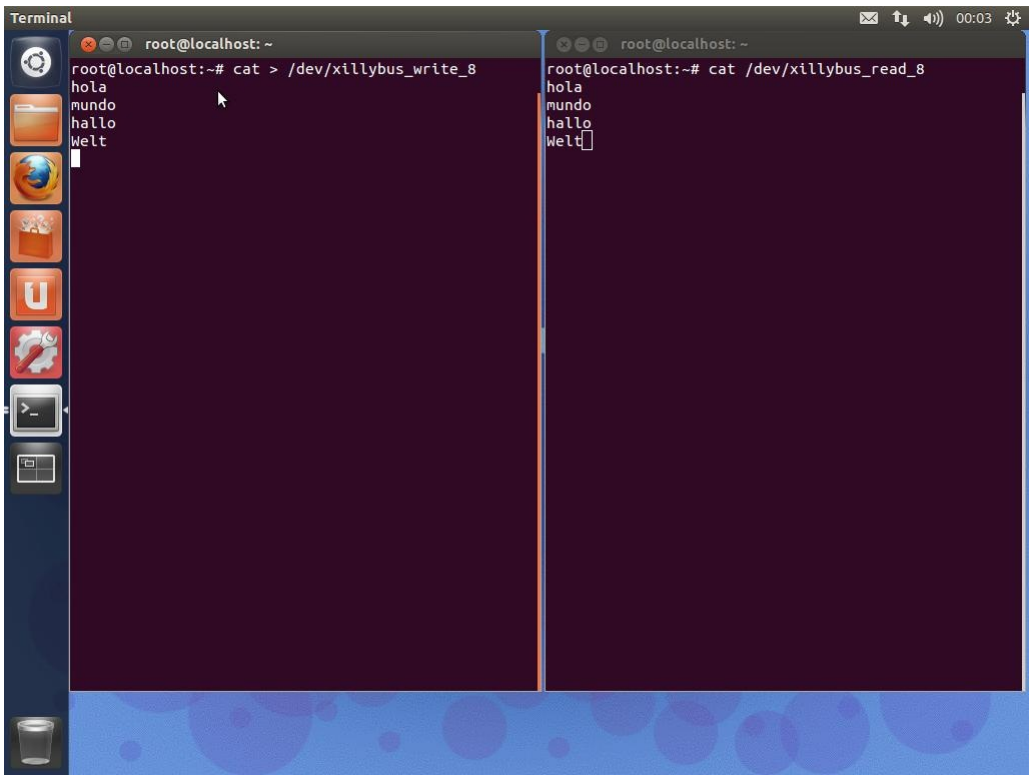

Figure 3.5: Two terminals, one writes the data in the FIFO and the other receives the data.

To get everything running we need to follow the steps mentioned in the paragraph 3.3:

- Modify the xillydemo.vhd file instantiating a new component of the fifo_8x2048 and connecting it properly to the previous FIFO and the Xillybus.
- Regenerate the bitstream file.
- Recreate the boot image file.
- Copy the new boot image file into the SD card.

In the next image we can see the test running properly.

This test has been made with the FIFO of 8 bits and the FIFO of 32 bits. No problem occurred during the tests.

To complicate a bit the loopback we also add some VHDL code that modifies the data between the FIFOs. This modification consists in adding 1 to the data that flows between the FIFOs.

As we receive the data in ASCII, if we add 1 to an A we get B, if we add 1 to a B we get C, and so on.

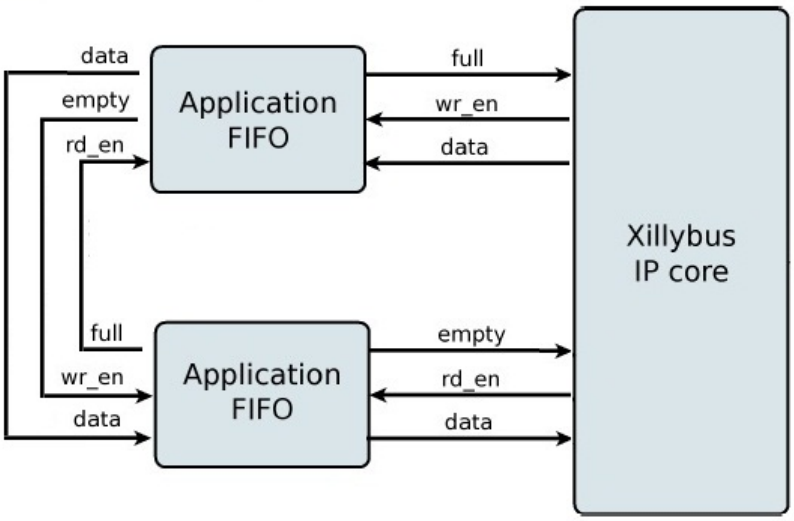

Figure 3.6: Adding a second FIFO in the loopback

In this point we will start adding our own custom logic to the previous project. Xillybus creators strongly recommend to not remove either substitute the FIFOs connected to the Xillybus, so the solution would be to connect our custom logic to the other side of one of its FIFOs, as we can see in the Figure 2.5.

3.3.4 AXI4 Stream FIFOs

The accelerator project is implemented with AXI4-Stream (AXI4-S) bus standard [5]. Due to this, as a first approach we will try to connect AXI4-Streams to the FIFOs loopback, in order to verify the correct connexion between the Native interface FIFOs and AXI4 interface. This action will also give us some feedback about the Xillybus' signals behaviour.

For this purpose, we will create a similar 8-bits FIFOs but with AXI4-Stream connections. We also used the Xilinx LogiCORE IP FIFO Generator[2], but selecting AXI4 interface (Figure 3.8).

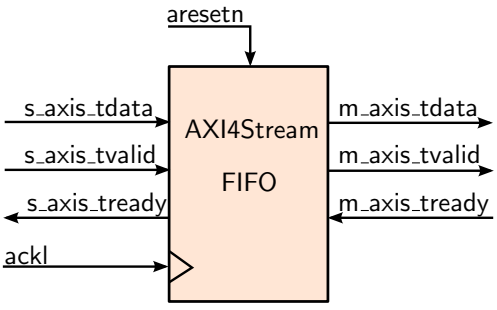

Figure 3.7: FIFO AXI4-Stream block.

Signal	Dir	Type	Description
ackl	in	std_logic	Clk input
aresetn	in	std_logic	Reset input (active low)
s_axis_tdata	in	std_logic_vector (8)	AXI4-S slave tdata input
s_axis_tvalid	in	std_logic	AXI4-S slave tvalid input
s_axis_tready	out	std_logic	AXI4-S slave tready output
m_axis_tdata	out	std_logic_vector (8)	AXI4-S master tdata output
m_axis_tvalid	out	std_logic	AXI4-S master tvalid output
m_axis_tready	in	std_logic	AXI4-S master tready input

Table 3.1: AXI4-S FIFOs input and output ports

The AXI4 interface protocol uses a two-way VALID and READY handshake mechanism. The information sources uses the VALID signal to show when valid data or control information is t with AXI4-Stream connections. We also used the Xilinx LogiCORE IP FIFO Generator[2], but selecting AXI4 interface (Figure 3.8).

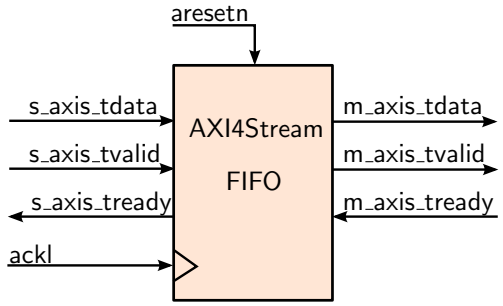

Figure 3.8: FIFO AXI4-Stream block.

available on the channel. The information destination uses the READY signal to show when it can accept the data. Figure 3.9 shows an example timing diagram for write and read operations to the AXI4 FIFO.

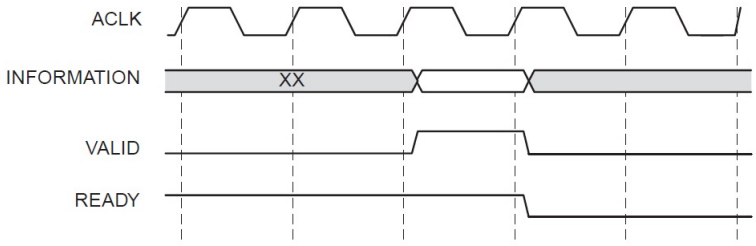

Figure 3.9: AXI4-FIFO Timing Diagram

In the standard FIFOs we do not have VALID either READY signals, instead of these we have WR_EN (write enable) and FULL connected to a write agent

and RD_EN (read enable) and EMPTY connected to a read agent. The side connected to the write agent will be the AXI4 slave side and the side connected to the read agent the AXI4 master side.

Firstly, proceed to analyze the slave part and the good connexion between the control signals. In the standard FIFO WR_EN is active when the write agent is ready to send data, so it basically has the same behavior as VALID. On the other hand, FULL means the FIFO is full, so it is not able to accept more data. This behavior is very similar to READY but inverted, so we will put a NOT gate in the middle of this connexion between FIFOs.

In the slave part we found the same behavior. RD_EN means the read agent is ready to receive the data, so it does the same as READY, and empty means the FIFO has no data to send, so is similar to VALID inverted.

The connection between FIFOs is based on paragraphs above. We can see the graphical connection in Figure 3.10.

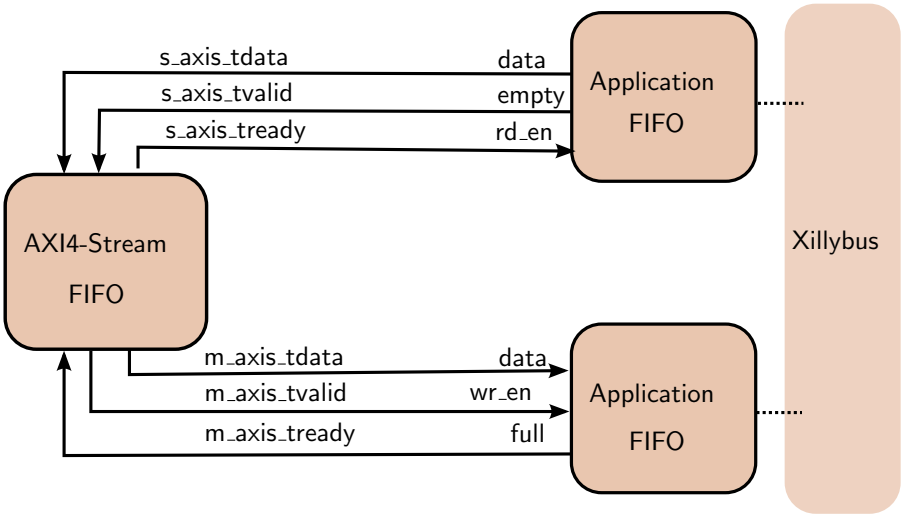

Figure 3.10: Connection between standard and AXI4-Stream FIFOs

To test our design we have created the AXI4-Stream FIFO with the *Xilinx Core Generator*, and later added the new files (ending in .v and .ngc) to our ISE proyect. After that we just need to modify the VHDL top module file adding the new component and connecting it. Afterwards we just follow the steps described before (Figure 3.4), introduce the SD card in the ZedBoard and turn on the board to test it.

We got satisfactory results, the same as we had only with the standard FIFOs, which means the connection is successful since it does not interfere in the original behaviour.

3.3.5 Integration with Serializer Interfaces

This step is crucial to the correct future implementation of the whole accelerator in our project. The accelerator works with an input of 580 bits and an output of 66, therefore we need an intermediate interface to transform the size of the input/output data vector. The solution for this is the use of a serializer interface developed in the research group by Luis Vega.

The serializer interface we will use need some changes before it fits in our project. The initial version has a different behavior than the one we need in our project. At this point, the correct performance of the AXI4-Stream signals is vital, if not we would not be able to correctly send the data from the Xillybus to the accelerator. The interface consist of two different cores, the first transforms serial data in parallel and the second transforms parallel data in serial. The serial-to-parallel transforms small data input in a bigger output (multiple of the input). Likewise, but in the other direction, the parallel-to-serial transforms big inputs in a smaller outputs.

Signal	Dir	Type	Description
G_RESET_ACTIVE	generic	std_logic	Specifies if the reset is active high (1) or low (0)
WIDTH	generic	integer	Bit width
NUM_REG	generic	integer	Number of registers
clk	in	std_logic	Clock signal, sensitive to positive clock edge
rst	in	std_logic	Reset input, depends on G_RESET_ACTIVE
s_axis_tvalid	in	std_logic	AXI4-S slave tvalid input
s_axis_tdata	in	std_logic_vector (WIDTH)	AXI4-S slave tdata input, depends on WIDTH value
s_axis_tready	out	std_logic	AXI4-S slave tready output
s_axis_tlast	in	std_logic	AXI4-S slave tlast input, not used
reg_array_ready	in	std_logic	Ready input, unit is able to write data if ready = 1
reg_array	out	std_logic_vector (WIDTH*NUM_REG)	Data output, depends on WIDTH and NUM_REG values
reg_array_valid	out	std_logic	If 1 data output is valid for its use

Table 3.2: Serial-to-parallel AXI4-S interface input and output ports

As we can see in the tables above, we can handle the behavior of the AXI4-S cores setting G_RESET_ACTIVE, WIDTH and NUM_REG values. For example, in the serial-to-parallel block we will set G_RESET_ACTIVE = 1, WIDTH = 8 and NUM_REG = 60, in order to get a block that works with a reset active high, an input data of 8 bits (to connect the 8 bit FIFO) and an output data of 480 bits (11 inputs of 32 bits each one and one more of 128). In the parallel-to-serial block we will have the same configuration with the exception of NUM_REG = 9, in this case we will have an input of 4 bits (2 outputs of 32 bits and 1 more of 2 bits, so we will only use the 66 more significants) and an output of 8 (to connect to the 8 bit FIFO).

The behavior of this blocks was modified from the original version since it does not work with the logic of the Xillybus. Both blocks have 3 states (Read, Load, Write). In the first state the blocks must read as much data as it needs, in the second they modify the data in order to transform serial-to-parallel or parallel-to-serial, and in the last one they write the transformed data in the output.

To emulate this block we connect it to the AXI4-S FIFOs we have created previously, the connexion is very simple as both blocks have AXI4-S interfaces. We connect the signals from the slave part to the master and vice versa.

At the first attempt, we kept the NUM_REG as 1, so the bit width in the input and output was the same. This first approach was very helpful, since we could debug almost all the VHDL code, finding the parts that were not working. Afterwards, we tried by modifying NUM_REG first to 3 and then to 9. In these

Signal	Dir	Type	Description
G_RESET_ACTIVE	generic	std_logic	Specifies if the reset is active high (1) or low (0)
WIDTH	generic	integer	Bit width
NUM_REG	generic	integer	Number of registers
clk	in	std_logic	Clock signal, sensitive to positive clock edge
rst	in	std_logic	Reset input, depends on G_RESET_ACTIVE
reg_array_ready	out	std_logic	Ready output, unit is able to write data if ready = 1
reg_array	in	std_logic_vector (WIDTH*NUM_REG)	Data input, depends on WIDTH and NUM_REG values
reg_array_valid m_axis_tvalid	in out	std_logic std_logic	If 1, data input is valid for its use AXI4-S master tvalid output
m_axis_tdata	out	std_logic_vector (WIDTH)	AXI4-S master tdata output, depends on WIDTH value
m_axis_tready	in	std_logic	AXI4-S master tready input
m_axis_tlast	out	std_logic	AXI4-S master tlast output, not used

Table 3.3: Parallel-to-serial AXI4-S interface input and output ports

cases, is a specification of our VHDL to send as many bits as number of registers we have, in other case, our interfaces will keep waiting for more data until they have enough to complete the output data vector.

Finally we were able to communicate our serializer interfaces with the Xillybus properly. To conclude, we just remove the AXI4-S FIFOs from the design and connect the serializer interfaces directly to the standard FIFOs.

3.4 Final Design

The accelerator design by Pedro consists in several IP cores, we will mention the most important ones and shortly describe them.

- Step Generator Core: This Block implements the Step Generating function of the Heston model. Internally it has a core generated with HLS and interfacing logic needed to control its behavior. It has two slave and one master AXI4-S interfaces. The current values of the price and volatility are fed through s_axis_input, whilst the Brownian increments through s_axis_rng.
- Correlator Core: The correlator is a block that takes two numbers and correlates them using a given Rho constant. This core comprises a multiplier, an adder, and the extra logic to store partial results. The arithmetic cores are implemented using the Xilinx Ip-Cores from the floating point library.
- Antithetic Core: The Antithetic unit comprises two FSM. The function of the block is to allow the run of simulations using antithetic numbers. This allows to have only a single RNG unit, but yet, have a throughput of two random numbers every clock cycle. This is done by negating the sign of the input numbers from the last clock cycle.
- Barrier Checker Core: The barrier checker is just a pair of digital comparators, and their aim is to check whether the input (the option's price) has reached a certain lower or upper bound. There are two comparators in order to check both barriers in parallel.
- Controller: This is a very simple controller implemented to synchronize the different cores of the datapath. It does not support flushing in the middle of the cycle, and it assumes that once the valid_conf flag is asserted, is not lowered during operation. The controller is located inside the Inner-Datapath.
- Accumulator Core: The correlator is a block that takes two numbers and correlates them using a given Rho constant. This core comprises a multiplier, an adder, and the extra logic to store partial results. The arithmetic cores are implemented using the Xilinx Ip-Cores from the floating point library.

After the correctness of each core, they were connected in order to form the complete design shown in Figure 3.1. Due to all the previous work and test, it should work with no problems and without the need of any change.

For a better understanding of the following table we need to explain that PQ means *packet queues* and RNG means *random number generator*.

Signal	Dir	Type	Description
aclk	in	std_logic	Clk input
aclken	in	std_logic	Clk enable
aresetn	in	std_logic	Reset input (active low)
rho	in	std_logic_vector (32)	Rho input for the correlator core
sqr_rho	in	std_logic_vector (32)	Square root of Rho input for the Correlator Core
valid_config	in	std_logic	Valid Configuration singal for the cores
antithetic	in	std_logic	Antithetic signal for the antithetic unit
m_axis_output_tvalid	out	std_logic	AXI4-S master tvalid output
m_axis_output_tready	in	std_logic	AXI4-S master tready input
m_axis_output_tlast	out	std_logic	AXI4-S master tlast output
m_axis_output_tdata_mx	out	std_logic_vector (32)	AXI4-S master output
m_axis_output_tdata_mv	out	std_logic_vector (32)	AXI4-S master output
m_axis_output_tdata_bar	out	std_logic_vector (2)	AXI4-S master output
s_axis_pq_tvalid	in	std_logic	AXI4-S slave tvalid PQ input
s_axis_pq_tready	out	std_logic	AXI4-S slave tready PQ output
s_axis_pq_tlast	in	std_logic	AXI4-S slave tlast PQ input
s_axis_pq_tdata	in	std_logic_vector (128)	AXI4-S slave tdata PQ input
s_axis_rng_tvalid	in	std_logic	AXI4-S slave tvalid RNG input
s_axis_rng_tready	out	std_logic	AXI4-S slave tready RNG output
s_axis_rng_tlast	in	std_logic	AXI4-S slave tlast RNG input
s_axis_rng_tdata	in	std_logic_vector (32)	AXI4-S slave tdata input with random numbers
kappa_delta	in	std_logic_vector (32)	Simulation parameter kappa delta
delta	in	std_logic_vector (32)	Simulation parameter delta
sqrt_delta	in	std_logic_vector (32)	Simulation parameter sqr_delta
risk_inter_rate	in	std_logic_vector (32)	Simulation parameter risk interest rate
long_term_avg_vol	in	std_logic_vector (32)	Simulation parameter long term average volatility
vol_of_vol	in	std_logic_vector (32)	Simulation parameter volatility of volatility
top_barrier_in	in	std_logic_vector (32)	Simulation parameter top barrier
bot_barrier_in	in	std_logic_vector (32)	Simulation parameter bottom barrier

Table 3.4: Accelerator input and output ports

The table above represents all the input and output signals of the complete accelerator. As we can see we have 12 input and 3 outputs, in total we need a input bit width of 480 bits and an output bit width of 66. The configuration of the serial-to-parallel interface needs to have the REG_NUM set in 60 and the parallel to serial interface set in 9. Therefore, our input and output bit will be setting with the follow configuration, this will be the order we must follow to

send the data properly.

Singal	Bit Location
s_axis_pq_tdata	352 to 459
rho	320 to 351
sqr_rho	288 to 319
s_axis_rng_tdata	256 to 287
kappa_delta	224 to 255
delta	192 to 223
sqr_delta	160 to 191
risk_inter_rate	128 to 159
long_term_avg_vol	96 to 127
vol_of_vol	64 to 95
top_barrier_in	32 to 63
bot_barrier_in	0 to 31

Table 3.5: Accelerator input data array

Singal	Bit Location
trash*	66 to 95
vol_of_vol	64 to 65
top_barrier_in	32 to 63
bot_barrier_in	0 to 31

Table 3.6: Accelerator output data array

In the output data array the data comprised between the 66 and 95 bits will be trash. As we don't have enough data to complete the whole array our parallel-to-serial interface will fill it with zeros.

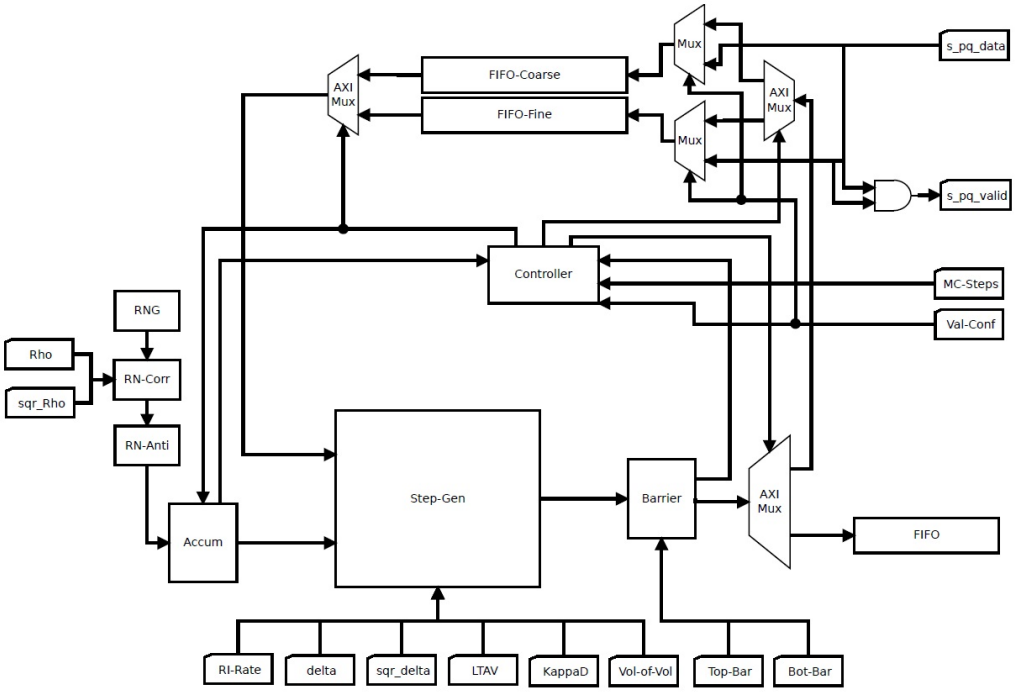

Figure 3.11: Block diagram of the complete accelerator

4. Results

In this chapter we will describe some practical results that we got from the implementation of the cores and the architecture described in chapter 3. Firstly we show the results of the complete architecture without the accelerator and after with it included. It was done in order to be able to test a simple architecture where maybe in the future we can add a more sophisticated accelerator.

4.1 First Architecture: Without the Accelerator

All the cores proposed in section 3 were successfully coded in VHDL and implemented. The previous ISE project from the Xillybus was modified in order to add our custom logic.

As mentioned earlier, in this architecture no accelerator was implemented in order to have a general design where we can add future more complex accelerators.

4.1.1 Resource Utilization

Table 4.1 shows resource utilization by this architecture. Synthesis and Place and Route options were the following:

- Resource Sharing: Activated.
- Shift Register Extraction: Activated.
- Optimization Goal: Speed.
- Optimization Effort: High.
- Place and Route mode: Only Route.
- Place and Route Effort Level: High.
- PAR extra effort: Normal.

Slice Logic Utilization	Used	Available	Utilization
Slice Registers	3.525	106.400	3%
Registers as FF	3.512		
Registers as AND/OR logics	13		
Slice LUT's	3.377	53.200	6%
LUT - FF pairs	4.287		
DSP48E1's	0	220	0%
BRAMs (36E1)	0	140	0%
BRAMs (18E1)	9	280	3%
Max Freq. [MHz]	111.707		

Table 4.1: Resource utilization without the accelerator.

Slice Logic Utilization	Used	Available	Utilization
Slice Registers	3.603	106.400	3%
Registers as FF	3.590		
Registers as AND/OR logics	13		
Slice LUT's	3.379	53.200	6%
LUT - FF pairs	4.466		
DSP48E1's	0	220	0%
BRAMs (36E1)	0	140	0%
BRAMs (18E1)	9	280	3%
Max Freq. [MHz]	111.707		

Table 4.2: Resource utilization with the accelerator

4.2 Second Architecture: With the Accelerator

As mentioned at the beginning of this chapter, in this architecture we included the actual accelerator to our design.

4.2.1 Resource Utilization

Table 4.2 shows resource utilization by this architecture. Synthesis and Place and Route options were the following:

- Resource Sharing: Activated.
- Shift Register Extraction: Activated.
- Optimization Goal: Speed.
- Optimization Effort: High.
- Place and Route mode: Only Route.
- Place and Route Effort Level: High.
- PAR extra effort: Normal.

4.3 Trade-offs of the design flow

In this work we developed a first approximation of a solution to eliminate the need of an external host. In consequence, the communication problem between an external host processor (ARM processor) and the desired hardware accelerator (FPGA) are slashed. Moreover, we also improve the communication speed as everything is embedded in the ZedBoard.

Nevertheless, another solution would be "peer processing". "Peer processing" means that we have two process unit within the same layer or hierarchy level. In this case there is no need of an OS, the communication between the PS and the PL is done through memory. We promote this as an alternative solution because of the ARM Cortex-A9[15].

The ARM Cortex-A9 processor is combined with a rich set of embedded peripherals, interfaces, and on-chip memories to create a complete hard processor system (HPS). The high-bandwidth on-chip backbone connecting the HPS and FPGA fabric provides over 100 Gbps peak bandwidth, ideal for sharing data between the ARM processor and hardware accelerators within the FPGA fabric.

The "peer processing" is part of a new design direction, which has the FPGA as part of a CPU systems which represents an asymmetric multi-processing capability with extensible I/O, high speed serial transceivers, and an integrated micro-controller in addition to the main multi-processing cores. These systems can connect can connect to outside high performance systems such as QPI and hyper-transport interfaces.

Summarizing, our soltion has good results for a less design effort instead of the "peer processing". However, this other solution could exploit the Zynq resources better.

4.4 Future Work

As a future line of work, we see two main fields. The first one is to implement a brand new design in "peer processing", and the second to improve our design. For the second one we would like to leave a record of those specific details that we think could improve future designs:

- Remove the FIFOs. Remove the standard FIFOs from our design analyzing in a lower level Xillybus architecture.
- Create our custom IP Xillybus core. Our own IP core bus will reduce its design just letting the parts we are interested in. It would minimize the power and source comsumption and therefore, increase the speed.

4.5 Conclusion

The embedded systems is growing faster and faster everyday. Nowadays, the kind of things we can accomplish in an embedded environment will be much more complex than they were 10 years ago. Being able to provide an environment that is secure and highly available while still delivering deterministic real-time characteristics is very important.

Therefore, is getting more difficult to overcome nowadays implementation challenges and yet meet the always increasing demands in computing power. For this an end-to-end optimization of all the layers of the system is needed.

In our work we had the opportunity to take a step towards this idea, by implementing, to the best of our knowledge, the first migration of the hardware accelerator to the ZedBoard for the purpose of opening the doors to the final implementation of a *hardware-software-co-design flow*.

We experienced the advantages and disadvantaged of FPGA based designs, being the main advantage of such platforms the reconfigurability, but having as a counterpart the design effort, which is still very high.

We acknowledge the advantages and the potential of the migration of embedded systems concepts into the HPC field, however further developments are needed in order to further automate the implementation stage and to provide a feasible time-to-market. The acceleration of financial algorithms is a booming research field, and it is a very interesting playground that could be used to develop design paradigms that can be used in other areas of HPC such as Physics, Biology, Geophysics, etc.

Bibliography

[1] Xilinx (July 23, 2010). *LogiCORE IP FIFO Generator v6.2 User Guide* UG175.

[2] Xilinx (July 25, 2012). *LogiCORE IP FIFO Generator v9.2 Product Guide* PG057.

[3] Pedro M. Torruella Naranjo, *"FPGA Based Multi-Level Monte-Carlo Hardware Accelerator for the Heston Model: an implementation proposal"*. Master Thesis in the Microelectronic System Design Research Group, University of Kaiserslautern.

[4] Primozic, T. (2011).*Estimating expected first passage times using multilevel monte carlo algorithm*. Master's thesis, New College, University of Oxford.

[5] Xillinx (2011). *AXI Reference Guide*. Xilinx Inc., v13.1 edition.

[6] Michael B. Giles, *"Multilevel Monte Carlo Path Simulation"*, Operations Research Journal, 2008, Vol. 56, no. 3, pp. 607-617.

[7] S. Heinrich. Multilevel Monte Carlo Methods, volume 2179 of Lecture Notes in Computer Science, pages 58-67. Springer-Verlag, 2001

[8] N. Metropolis, S. Ulam, *"The beginning of the Monte Carlo method"*, Journal of the American Statistical Association (American Statistical Association), 1949-Sept, Vol. 44 No. 247, pp. 335341

[9] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, R. Korn, *"An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model"*, in Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, Nov. 30 2011-Dec. 2 2011, pp. 468-474 .

[10] Fischer Black and Myron Scholes, *"The Pricing of Options and Corporate Liabilities"*, The Journal of Political Economy, Vol. 81, No. 3 (May - Jun 1973), pp. 637-654.

[11] Robert C. Merton, *"Theory of rational Option Pricing"*, The Bell Journal of Economics and Management Science, Vol. 4, No. 1 (Spring, 1973), pp. 141-183.

[12] de Schryver, C., Marxen, H., & Schmidt, D. (2011a). Hardware Accelerators for Financial Mathematics - Methodology, Results and Benchmarking. In Young Researchers Symposium (YRS) 2011, Proceedings on (pp. 5560).: Center for Mathematical and Computational Modelling (CM)2; (CM)2; Nachwuchsring.

[13] de Schryver, C., Shcherbakov, I., Kienle, F., Wehn, N., Marxen, H., Kostiuk, A., & Korn, R. (2011b). An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs, RECONFIG 11 (pp. 468474). Washington, DC, USA: IEEE Computer Society.

[14] Christopher Z. Mooney, Monte Carlo Simulation *Serie: Quantitative Applications in the Social Sciences*, a SAGE UNIVERSITY PAPER 1997

[15] ARM 2009. White paper, *The ARM Cortex-A9 Processors*

Acronyms

IP Intelectual Property

ISE Integrated Software Environment

XPS Xilinx Platform Studio

FPGA Field Programmable Gate Array

MC Monte Carlo

MLMC Multi-Level Monte Carlo

PS Processing System

PL Programmable Logic

EPP Extensible Processing Platform

ASSP Application Specific Standard Product

FIFO First In First Out

GPL General Public License

CPU Central Processing Unit

HPC High Performance Computing